OpenGL
What Is OpenGL?

- A software interface (API) to the graphics card.
- Client-Server abstraction
  - The client is your program, it sends rendering commands to the server.
  - The server is the graphics card – get the commands, interpret them and produces pixels on screen.
- Pipeline oriented abstraction of the rendering process.
- Extremely portable – supported by most OS and display cards.
- Based on algorithms that were developed by Silicon Graphics, Inc (SGI).
  - T&L
  - Texture mapping
  - Alpha blending
What Is OpenGL - Cont

- Common Algorithms are implemented in HW.
- There is an international consortium of companies (ARB) that decides on future features
  - Most of the big names are represented
    - IBM, Microsoft, SGI, HP, ATI, Nvidia, 3dLabs ....
- A procedural (interpreter like) rather than a descriptive (vrml like) graphics language.
- Two modes of operation
  - Immediate – only rendering context is managed on card. we concentrate on immediate mode, simpler.
  - Retained – the geometry is moved in one piece, and managed on the graphics card.
OpenGL – history and feature list

- Version 1.0 – 1992
  - Basic geometry primitives, flat and Gauroud shading, Parametric textures, alpha and stencil buffers and more
- Version 1.1 -1996
  - Vertex Arrays, Texture enhancements
- Version 1.2 – 1998
  - Extension mechanism, 3D textures, RGBA pixel format
- Version 1.3 – 2001
  - Compressed textures, Cube-maps, multitextures, multisamples
- Version 1.4 – 2002
  - Automatic MipMap, Fog coordinates,
- Version 1.5 – 2003
  - Vertex Buffers, Occlusion queries, Full set of Boolean operators
- Version 2.0 – 2004
  - OpenGL Shading Language (glsl) is the most notable addition
- Version 2.1 – 2006
  - PBOs, sRGB, non-square mtx
- Version 3.0 /3.2/3.2– (2008-2010
  - GLSL 1.3-1.5, Geometry shaders
- Version 4.0-4.1 – 2010 – Equal to DX11
  - GLSL 4.0, OpenCL support, OpenGL ES
Modern OpenGL

- Used mainly in CAD and embedded (cell phones)
How OpenGL Works?

- The client describes the necessary steps to achieve a certain appearance or effect.
  - A stream of vertices is “pushed” to the card.
    - Along with connectivity information
  - Each vertex is transformed and lit (T&L phase).
  - The result primitive is scan-converted and generate fragments (that might become screen pixels)
  - Z-Buffer is applied to the fragments (also stencil/alpha testing).
- The fragments that pass all tests are textured and displayed
Geometric Primitives

- The following geometric primitives in 3D may be used:
  - Points
  - Lines
  - Polygons
  - Triangle strips/fans
  - Quads

- Also supports:
  - Lighting
  - Shading
  - Texture mapping
  - Animation
  - Other special effects

- Does not include any functions for:
  - Windows management
  - User interaction
  - File I/O
MFC with OpenGL

- Open a window
  - Can be done using MFC or plain Win32
  - Glut, fltk, gtk all have GL display classes/procedures

- OnCreate function of the view
  - Fill out the PIXELFORMATDESCRIPTOR structure.
    - Describe the graphics setup.
  - Pass this structure to the ChoosePixelFormat() API.
    - It returns an integer index to an available pixel format for the specified device context.
    - Pass this index to SetPixelFormat() function.
  - Create a rendering context
    - wglCreateContext() –
      - Every function starting with wgl* is win32 specific
      - There xgl* equivalent for most.
  - Make it current
    - wglMakeCurrent()
MFC with OpenGL

- **OnDestroy** function of the view
  - The GL DB must be cleared at the end using
    - `wglDeleteContext()`

- Regular use
  - After issuing an OpenGL command check the status with
    - `glGetError()`
    - If different from GL_NO_ERROR debug.
Double Buffering

- Most graphics applications do not wish to have the flickering effect resulting from window updates.
  - This is solved by having two buffers for a device context.
  - We always update the back buffer.
  - Once it’s ready, we switch the two buffers, starting the update again.

- Windows has support for double buffering using the GDI function
  - `BOOL SwapBuffers( HDC hdc);`
OpenGL supports ten primitives

- **GL_POINTS**: Individual points
- **GL_LINES**: Vertices pairs, individual line segments
- **GL_LINE_STRIP**
- **GL_LINE_LOOP**
- **GL_TRIANGLES**: Vertices triplets interpreted as triangles
- **GL_TRIANGLE_STRIP**
- **GL_TRIANGLE_FAN**
- **GL_QUADS**: Vertices quadruples, 4 sided polygons
- **GL_QUAD_STRIP**
- **GL_POLYGON**: Boundary of a simple, convex polygon
GL Primitives (cont.)

GL_POINTS

GL_LINES

GL_TRIANGLES

GL_POLYGON
- A Primitive is started by
  - `glBegin(GLenum mode);`
- And ended using
  - `glEnd();`
- Between them you can specify the primitive coordinates, color, normal etc.
- For example:

```c
glBegin(GL_LINES);
  glVertex3f(0.0, 0.0, 0.0);
  glVertex3f(100.0, 100.0, 0.0);
  glVertex3f(100.0, 0.0, 0.0);
  glVertex3f(0.0, 100.0, 0.0);
glEnd();
```
Specifying Vertices:

- `void glVertex{234}{sifd}[v](TYPE coords);`
  
  (for \{x, y, z, w\}) and short, int, float or double

- Example:

  \begin{verbatim}
  glVertex2s(2, 3);
  glVertex3d(0.0, 0.0, 3.1415927);
  glVertex4f(2.3, 1.0, -2.2, 2.0);
  \end{verbatim}

  \begin{verbatim}
  GLdouble dvect[3] = \{5.0, 9.0, 1992.0\};
  glVertex3dv(dvect);
  \end{verbatim}

- Normals

  - `Void glNormal3{bdfis}[v]{params}`

- Colors

  - `Void glColor{34}[u]{bdfis}[v]`
OpenGL Options

- The `glEnable()` and `glDisable()` functions enable or disable OpenGL capabilities.

  - GL_DEPTH_TEST for z-buffer, hidden surface elimination
  - GL_CULL_FACE for enabling back face culling
  - GL_LIGHTING for lightning enabling
  - GL_LIGHTi
  - GL_DITHER for color dithering
  - GL_BLEND
  - GL_COLOR_MATERIAL

  Etc…
Stages of Vertex Transformations

\[ \begin{bmatrix} X \\ Y \\ Z \\ w \end{bmatrix} \rightarrow \text{Modelview Matrix} \rightarrow \text{Projection Matrix} \rightarrow \text{Viewport Transformation} \rightarrow \text{clipped coordinates} \]

- **eye coordinates**
- **perspective division**
- **normalized device coordinates**
- **window coordinates**
The ModelView Matrix

- A 4x4 matrix M representing the transformed coordinate system.
- Setting the matrix mode to modelview
  - `glMatrixMode(GL_MODELVIEW);`
- Each vertex V you provide is a single vector matrix.
- Before the rendering process, Each V is multiplied by the Modelview matrix.
  - \( V' = M \times V \)
The Projection Matrix

- A 4x4 matrix M representing the projection matrix.
- Setting the matrix mode to projection
  - `glMatrixMode(GL_PROJECTION);`
  - Defines the clipping volume – object out of the clipping volume will be culled.

- Initializing functions
  - `void glOrtho( GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far );`
  - `void glFrustum( GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble znear, GLdouble zfar );`
  - `void gluPerspective( GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar );`
Transformations

- Rotation
  - \texttt{glRotated(GLdouble angle, GLdouble x, GLdouble y, GLdouble z)}

- Scaling
  - \texttt{glScaled(GLdouble x, GLdouble y, GLdouble z)};

- Translation
  - \texttt{void glTranslated(GLdouble x, GLdouble y, GLdouble z)};

- The \texttt{active} GL matrix \( M \) is updated with the transformation \( T \) as follows:
  - \( M = M \times T \)

- Each transformation effects only the vertices defined \texttt{after} the transformation call.

- In general, you make the active matrix the Modelview matrix.
Other Matrix Operations

- Selecting the matrix mode is done using
  
  ```c
  void glMatrixMode( GLenum mode );
  ```

- `mode` can be one of
  - GL_MODELVIEW
  - GL_PROJECTION
  - GL_TEXTURE
Matrix Stack

- The active matrix is saved on a stack.
  - The top of the stack is the active matrix
  - `glPushMatrix();`
  - `glPopMatrix();`

```c
void drawCar() {
    draw_car_body();
    glPushMatrix();
    glTranslatef(40, 0, 30); // modeling – move the wheel
    draw_wheel();
    glPopMatrix();
}
```
To initialize an identity matrix use
- `glLoadIdentity()`

To load a given matrix use
- `void glLoadMatrix{d|f}( const GLdouble *m );`
  - `m` is a pointer to a 4x4 matrix stored in column-major order as 16 consecutive values

In order to directly multiply the active matrix with a given matrix, use
- `void glMultMatrix{d|f}( const GLdouble *m );`
  - `m` is a pointer to a 4x4 matrix stored in column-major order as 16 consecutive values
Accessing GL State

- `glGetDoublev(GLenum pname, GLdouble * params);`

  ■ `pname` may be
    - `GL_COLOR_CLEAR_VALUE` (background color)
    - `GL_CURRENT_COLOR` (current paint color)
    - `GL_CURRENT_NORMAL` (current normal)
    - `GL_MATRIX_MODE`
    - `GL_MODELVIEW_MATRIX`
    - `GL_PROJECTION_MATRIX`
    - `Etc.`

  ■ The size of the returned `params` array depends on the `pname` used.

  ■ Minimize the use of this call – it is expensive.
Normals

- Specifying the normal
  - `void glNormal3d(GLdouble nx, GLdouble ny, GLdouble nz);`
- When using lighting effects, normals must be defined for each polygon or for each vertex. OpenGL does not calculate a normal by itself.
- Note that `glScale` affects normals as well as vertices
  - `glEnable(GL_NORMALIZE)`
    - Automatically normalizes the given normals
Shading Models

- Choose the shading model
  - `void glShadeModel( GLenum mode );`
  - mode may be
    - `GL_FLAT`
    - `GL_SMOOTH` (the same as the Gouraud shading model)

- OpenGL does not support the Phong shading model
  - Programmable graphics

- Note that lighting must be enabled
  - `glEnable(GL_LIGHTING)`
Material Properties

- An object has certain reflective properties
- When using lighting, each model part has its own material properties, which also define its color.
  - \textit{glMaterialfv(GLenum face, GLenum pname, const GLfloat *params)}
  - \textit{Face} must be one of \textit{GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK}.
  - \textit{pname} can be \textit{GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_SHININESS} etc.
  - \textit{params} change accordingly.
Colors

In case you want to specify colors per face or per vertex, you must

- Enable color tracking.
  - `glEnable(GL_COLOR_MATERIAL);`
- Set material properties to follow `glColor` values.
  - `glColorMaterial(GLenum face, GLenum mode);`
- Specify the color before defining it’s corresponding vertex.
  - `glColor3d(GLdouble red, GLdouble green, GLdouble blue);`
  - `glColor3ub(GLubyte red, GLubyte green, GLubyte blue);`
Adding Light to a Scene

- Enabling lighting
  - `glEnable(GL_LIGHTING);`

- OpenGL supports up to `GL_MAX_LIGHTS` lights simultaneously
  - Query with `glGetIntegerv(GL_MAX_LIGHTS)`

- Set the light properties
  - `void glLightfv(GLenum light, GLenum pname, const GLfloat *params );`
  - `pname` can be `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_POSITION` etc.
    - `params` change accordingly.

- Enable the specific light
  - `glEnable(GL_LIGHTi);`
  - `i` is between 0 and `GL_MAX_LIGHTS-1`
Work Process

- Initialize the projection matrix.
- Make sure all the GL options you wish to use are on.
- Initialize any effects you wish to use, such as lights, fog, etc.
- For each object you work with in the scene:
  - Initialize the object material properties.
  - Initialize the transformation matrix.
  - Initialize all the primitives you wish to use (for example, all the polygons in the scene).
- Call glFlush() to apply all actions on the depth buffer.
OpenGl utility library (GLU)

- Some of the more frequent functions were not considered OpenGl “core”
  - Modeling support – cylinders, disks, NURBS curves and surfaces.
  - Helper function to generate transformation matrix – gluProject, gluLookAt, gluOrtho2D
  - Tessellate support – creating primitives out of high order surfaces.
  - Picking support functions
  - Mipmap support – build a hierarchy of sampled textures for different level of details, cost trice the memory and reduces aliasing effects.
Glx/Glw

- Glx library – work in the x-windows environment to help managing the open gl app
  - Double buffer
  - Context create and delete
  - Event handling

- Glw – similar functionality for windows